



# Fundamentos de Programación

Jesús Carretero Pérez  
Félix García Carballeira  
José Manuel Pérez Lobato  
Javier Fernández Muñoz  
Alejandro Calderón Mateos

Administración de Sistemas Informáticos

# INFORMÁTICA

## ACLARACIÓN

En este libro se utilizan las instrucciones `scanf` y `printf` (pertenecientes a C standard) para la entrada y salida de datos. Sin embargo, durante la asignatura, utilizaremos las instrucciones `cin` y `cout` (pertenecientes a C++) para este cometido. En la siguiente tabla tenemos las equivalencias:

	C standard	C++
Entrada	<code>scanf ("%formato", variable);</code>	<code>cin &gt;&gt; variable;</code>
Salida	<code>printf ("%formato", &amp;variable);</code>	<code>cout &lt;&lt; variable;</code>

### Ejemplos

<code>printf ("hola mundo %d", X);</code> <code>scanf ("%d", &amp;Y);</code>	<code>cout &lt;&lt; "hola mundo" &lt;&lt; X;</code> <code>cin &gt;&gt; Y;</code>
---	---

En la programación estructurada las aplicaciones tienen una zona en la que se definen los datos que se van a utilizar en el algoritmo y la zona de las instrucciones que los van a utilizar (procesar). Cuando el programador piensa en el algoritmo lo hace tal y como hemos indicado en el ejemplo del apartado anterior: centrándose en las instrucciones que van a formar parte del algoritmo y en los «lugares» en los que se va a almacenar la información que se necesita.

En la programación orientada a objetos el planteamiento inicial es completamente diferente pues, en este caso, en lo primero que hay que pensar es en los objetos o elementos que van a participar en la aplicación. Es decir, si quiero hacer un programa para gestionar un banco con programación orientada a objetos, tendré que pensar en cuáles son los objetos que van a participar: Clientes, Cuentas Bancarias, Sucursales, Empleados, Cajeros Automáticos, etc., mientras que si lo planteara con programación estructurada me centraría más en los procedimientos: tengo que dar de alta cuentas, pagar a los empleados, contabilizar el dinero entregado en los cajeros, etc.

En este libro nos vamos a centrar en la programación estructurada, puesto que C es un lenguaje de este tipo.

## 2.4 Elementos básicos de un programa

Los elementos básicos de los que consta un programa son:

- los datos;
- las instrucciones que los procesan.



### Propiedades de una variable:

- nombre;
- tipo de dato que almacena;
- valor que almacena.

### 2.4.1 Los datos

Para almacenar los datos necesitamos zonas de la memoria del ordenador. La memoria del ordenador está dividida en casillas numeradas en las que se guardan datos almacenados en sistema binario, en el que sólo se usan ceros y unos. Para utilizarla directamente tendríamos que tener un control riguroso sobre qué casillas hemos usado y sobre qué procedimiento hemos utilizado para convertir la información numérica o alfabética en estos ceros y unos. Afortunadamente, los lenguajes de programación nos permiten:

- asignar nombres simbólicos a estas casillas de tal forma que yo le pueda indicar al ordenador que quiero que una casilla se llame Apellidos, otra Nombre, otra Teléfono, etc.;
- controlar qué zonas de la memoria están libres y cuáles están ocupadas, de forma transparente y sin intervención del programador;
- convertir la información numérica o alfabética que usamos nosotros en una codificación que entienda el ordenador y viceversa.

Gracias a esto nosotros simplemente tenemos que indicarle al ordenador cómo queremos que se llamen las zonas de memoria donde vamos a almacenar información y qué tipos de datos van a almacenar, y él se ocupará de todo lo demás.

A estas zonas de memoria las llamaremos variables y cuando se definan se deben indicar los siguientes aspectos de las mismas:

- Nombre: que sirve para identificarla. El nombre de un elemento se formará con letras o números, empezando siempre con una letra; por ejemplo Domicilio, Numero1, PI. No serán válidos nombres como 213, 2<sup>a</sup>, a+b, etc. Los lenguajes de programación no admiten que se coloquen acentos en los nombres de las variables.
- Tipo: indica qué conjunto de valores puede almacenar la variable. Los tipos básicos son:
  - número entero: -500, 0, 876;
  - número real: 4.5, 100.0, 12.56 \* 103;
  - carácter: una letra, un número de un dígito o un símbolo. Se suelen colocar entre comillas, por ejemplo: 'A', '4' , ':' , '!';
  - cadena de caracteres: varios caracteres, por ejemplo «José Manuel Pérez Lobato»;
  - lógico: sólo puede tomar los valores verdadero y falso.

- Valor almacenado.

Las variables nos permitirán almacenar un valor, que podremos sustituir por otro diferente durante la ejecución del programa tantas veces como queramos. Por ejemplo, en la variable N1 podemos inicialmente almacenar el número 1, después podemos utilizarla para almacenar el 2, posteriormente podemos hacer que almacene el número 56 en lugar del 2, etc.

Para almacenar un valor en una variable se utilizan las instrucciones de asignación que veremos a continuación.

## 2.4.2 Las instrucciones

Una de las instrucciones más utilizada en un programa es la de asignación, que tiene el formato:

**Variable ← Expresión**

El símbolo que se utiliza depende del lenguaje de programación. En lenguaje C se utiliza el símbolo = pero de momento utilizaremos el símbolo ← porque refleja mejor el sentido de esta instrucción.

En la instrucción de asignación se coloca a la izquierda del símbolo de asignación la variable a la que se le va a colocar el valor y a la derecha una expresión que puede ser simplemente un dato (numérico o de otro tipo) o una operación más compleja (como sumas, raíces cuadradas, etc.) Si la parte derecha es una expresión, lo primero que hace el ordenador es calcular su valor realizando las operaciones precisas; una vez obtenido el resultado de la expresión, éste es asignado a la variable.

Cuando se almacena un valor en una variable se pierde el valor que tenía almacenado anteriormente, pues éste es sustituido por el nuevo valor.

Veamos algunos ejemplos:

**dia ← 25** se coloca el valor 25 en la variable dia.

**sueldo ← 1200 + 1200\*0.1** se coloca el valor 1320 (resultado de la operación) en la variable sueldo.



Formato de la instrucción de asignación:  
**Var ← Expresión**

Si en la parte derecha de la asignación aparece el nombre de una variable, se sustituirá dicho nombre por el valor almacenado en la variable y después se calculará la expresión; por ejemplo:

`sueloJuan ← sueldo + 100` En este caso aparece a la derecha de la asignación la variable `sueldo` que tiene valor 1320 (de acuerdo con la instrucción del ejemplo anterior), por lo que la operación que se realizará será  $1320 + 100$ , con resultado 1420. Por tanto en la variable `sueloJuan` se almacenará el valor 1420.

`sueloAna ← sueldoJuan` En este caso a la derecha de la asignación aparece sólo la variable `sueloJuan` que tiene valor 1420 por tanto la asignación se transformaría en:

`sueloAna ← 1420` con lo que `sueloAna` también tendría el valor 1420.

Las expresiones se pueden utilizar, además de en la instrucción de asignación, en otras partes del programa. Los dos tipos más utilizados de expresiones son las numéricas y las condicionales.

### Expresiones numéricas

Las expresiones numéricas emplean los símbolos y siguen las reglas habituales de las matemáticas. Los operadores básicos son `+`, `-`, `*`, `/` y `%`, éste último calcula el resto de una división entera. Por ejemplo:

`A ← 3*5` almacena en A el valor 15;

`A ← 5/3` almacena en A el valor 1 (cociente de la división);

`A ← 5%3` almacena en A el valor 2 (resto de la división).

Si se van a utilizar operadores diferentes hay que tener en cuenta cuál se va a evaluar primero. La expresión  $5 + 3 * 2$  puede dar como resultado 16 si se calcula primero la suma u 11 si se calcula primero la multiplicación. Si hay dudas lo mejor es colocar paréntesis. En general se evalúan primero los operadores de multiplicación y división y posteriormente los de suma y resta, por lo que la expresión anterior daría como resultado 11. Además, el cálculo se realiza de izquierda a derecha.

### Ejercicio resuelto 2.2

Indicar el valor almacenado en cada variable después de la ejecución de las siguientes instrucciones, suponiendo que sirven para almacenar números enteros.

SOLUCIÓN.

`A ← 2;`      A almacena 2

`B ← A+1;`      B almacena 3

`A ← A+B*2;`      A almacena 8

`A ← A%B;`      A almacena 2

### Ejercicio propuesto 2.2

Indicar el valor almacenado en cada variable después de la ejecución de las siguientes instrucciones, suponiendo que sirven para almacenar números enteros:

`A ← 1`

`B ← A*2+A`

`A ← 2*(A+B*2)`

`A ← A%B`

`B ← B/A`



Los operadores aritméticos básicos son:

- `+` suma;
- `-` resta;
- `*` multiplicación;
- `/` división, si los operandos son números reales y cociente de división entera, si los operandos son números enteros;
- `%` resto de la división entre números enteros.

## Expresiones condicionales o lógicas

Expresiones condicionales o lógicas son aquéllas cuyo resultado es verdadero (V) o falso (F), que son los valores lógicos o booleanos. Los operadores que se utilizan son de dos tipos:

- operadores de comparación, y
- operadores lógicos.

Los operadores **relacionales o de comparación** permiten comparar el valor de dos variables o datos según varios criterios. Se utilizan colocando a la derecha del operador un valor, nombre de variable o expresión y a la izquierda otro y devuelven como resultado verdadero o falso.

- **operando1 > operando2**

Da como resultado verdadero si el operando1 es mayor que el operando2 y falso en caso contrario.

- **operando1 < operando2**

Da verdadero si el operando1 es menor que el operando2 y falso en caso contrario.

- **operando1 >= operando2**

Da verdadero si el operando1 es mayor o igual que el operando2 y falso en caso contrario.

- **operando1 <= operando2**

Da verdadero si el operando1 es menor o igual que el operando2 y falso en caso contrario.

- **operando1 == operando2**

Da verdadero si ambos operandos tienen el mismo valor y falso en caso contrario.

- **operando1 != operando2**

Da verdadero si ambos operandos tienen distinto valor y falso en caso contrario.



Los operadores de comparación básicos son:

- > mayor que;
- < menor que;
- >= mayor o igual que;
- <= menor o igual que;
- == igual a;
- != diferente de.

## Ejercicio resuelto 2.3

Indicar los valores de las expresiones suponiendo los valores siguientes almacenados en las variables.

A1 ← 30

A2 ← 20

**SOLUCIÓN.**

5 > 4	Verdadero
A1 < 5	Falso
A1 <= A2	Falso
A1-10 == A2	Verdadero
A1*2 == A2*3	Verdadero
A1 != 30	Falso

### Ejercicio propuesto 2.3

Indicar los valores de las expresiones suponiendo los valores siguientes almacenados en las variables:

A1 ← 30

A2 ← 20

A1 >= A1

A2 > A2

A1 > A2+14-A1

Los operadores lógicos se utilizan cuando se necesitan expresiones condicionales más complejas. Hay operadores que necesitan dos operandos y operadores que requieren un único operando.

- Para utilizar un operador lógico con dos operandos se coloca a la izquierda del mismo una variable, valor o expresión cuyo resultado sea verdadero o falso y a la derecha otra.
- Para utilizar un operador lógico con un operando se coloca a la derecha del mismo una variable, valor o expresión cuyo resultado sea verdadero o falso.



Los operadores lógicos básicos son:

- **&&** AND, operación Y;
- **||** OR, operación O;
- **!** NOT, operación NO.

Los operadores lógicos básicos son:

- **operandoLogico1 && operandoLogico2**  
Operador AND o Y, da como resultado verdadero si ambos operandos lógicos son verdaderos y falso en cualquier otro caso.
- **operandoLogico1 || operandoLogico2**  
Operador OR o O, da como resultado falso si ambos operandos lógicos son falsos y verdadero en cualquier otro caso.
- **! operandoLogico**  
Operador NOT o NO, da como resultado verdadero si el operando es falso y falso si el operando es verdadero.

### Ejercicio resuelto 2.4

Indicar los valores de las expresiones suponiendo los valores siguientes almacenados en las variables:

A1 ← 30

A2 ← 20

SOLUCIÓN.

A1>5 && A1<40

Verdadero, pues ambas comparaciones son verdaderas;

A1>5 && A1<30

Falso. La segunda comparación es falsa y al utilizar el operador && el resultado final es falso, pues una de ellas es falsa;

A1>5 || A1<30

Verdadero, con que la primera condición sea verdadera es suficiente;

A1>A2 && A2<15 && A2>10

Falso, pues A2<5 es falso;

A1>A2 || A2<15 || A2>10

Verdadero.

Respecto a la precedencia de operadores (cuáles se evalúan primero), en C se sigue el siguiente criterio:

- 1º ! (NOT)
- 2º >, >, >=, <=
- 3º ==, !=
- 4º && (AND)
- 5º || (OR)

Si hay dudas, lo mejor es colocar paréntesis.



Prioridad de los operadores (cuál se evalúa primero):

1. Paréntesis ()
2. !
3. \*, /, %
4. +, -
5. <, >, >=, <=
6. ==, !=
7. &&
8. ||

### Ejercicio resuelto 2.5

Indicar los valores de las expresiones suponiendo los valores siguientes almacenados en las variables:

A1 ← 30

A2 ← 20

SOLUCIÓN.

$A2 < 15 \&\& A1 > A2 \mid\mid A2 > 10$

Verdadero, pues tenemos F&&V || V. Al evaluar primero el && obtenemos F || V, que es verdadero.

$A2 < 15 \&\& (A1 > A2 \mid\mid A2 > 10)$

Falso, pues tenemos F&&(V || V). Al evaluar primero los paréntesis queda F&&V, que es falso.

### Ejercicio propuesto 2.4

Indicar los valores de las expresiones suponiendo los valores siguientes almacenados en las variables:

A1 ← 30

A2 ← 20

$A1 > 5 \&\& A1 != 44 \&\& A2 + A1 > A2 * 2$   
 $(A1 > 5 \mid\mid A2 < 10) \&\& A1 < 40$   
 $A1 > 3 \mid\mid A2 == 20 \&\& A2 == A1$

### Ejercicio resuelto 2.6

Las expresiones condicionales sirven para saber si el valor de una o varias variables cumple determinados criterios. Realizar una expresión condicional que sea verdadera cuando el valor de la variable A esté comprendido entre 1 y 10, incluidos el 1 y el 10.

SOLUCIÓN.

$A >= 1 \&\& A <= 10$

### Ejercicio propuesto 2.5

Realizar dos expresiones condicionales diferentes que sean verdaderas cuando el valor de la variable A no tome valores del 1 al 10, ambos incluidos.

## 4.1 Los tipos de datos

Todos los programas informáticos manejan datos, pero ¿qué es un dato? Se puede definir un dato como un «hecho o valor a partir del cual se puede inferir una conclusión; información». Por tanto, se puede concluir que todo aquello que un programa manipula son datos y que sin datos un programa no funcionaría correctamente. De hecho, las propias instrucciones de más bajo nivel del computador se basan en parámetros de entrada, operación sobre ellos y generación de salida. Este mismo modelo se puede aplicar a alto nivel cuando se procesa un fichero de entrada y se obtiene un fichero de salida, como se muestra en la Figura 4.1. Además, para ello se usan datos internos que, generalmente, son auxiliares para el procesamiento.

Ahora bien, no todos los datos son del mismo tipo y los programas manipulan datos de manera muy diferente según el tipo de dato del que se trate. Si se quiere manipular un entero, no se pueden usar operaciones de tipos reales. Por ello, en los programas es necesario poder distinguir tipos de datos. Pero ¿qué es un tipo de datos en un programa? Se puede establecer una definición muy sencilla de tipo extensivo: «El tipo de un dato es el conjunto de valores que puede tomar durante el programa». Si se le intenta dar un valor fuera del conjunto se producirá un error.



Los computadores manejan tipos de datos incluso al nivel más bajo. Por ejemplo, la UCP maneja internamente los registros o la palabra de memoria, tipos de datos característicos de la arquitectura de un computador.

¿Por qué es necesario asignar tipos a los datos? La asignación de tipos a los datos tiene cuatro objetivos principales. En primer lugar, determinar el conjunto de objetos que puede representar dichos datos. Por ejemplo, un tipo días del mes puede representar números naturales desde el 1 hasta el 31. En segundo lugar, conocer qué operaciones se pueden hacer sobre ese dato. ¿Se pueden sumar días del mes? Cada lenguaje de programación tiene unos tipos de datos definidos y unas operaciones sobre ellos. En tercer lugar, detectar errores en las operaciones, dado que cada tipo de datos tiene un comportamiento distinto en sus operaciones. Por ejemplo,  $3/4$  en enteros daría 0, pero en reales daría 0,75. Por último los tipos de datos permiten determinar cómo ejecutar estas operaciones. Por ejemplo, la multiplicación se ejecuta antes que la suma en  $3 * 4 + 5$ .

Existen muchas clasificaciones para los tipos de datos. Una muy sencilla es aquella que se hace teniendo en cuenta dos aspectos:

- Tamaño que ocupa en memoria fijo o variable: tipos estáticos y dinámicos.
- Número de elementos que representan: tipos básicos o complejos.

Un tipo de datos estático es aquél que ocupa un tamaño fijo en memoria, es decir, su tamaño no puede variar durante la ejecución del programa. Esto supone que una

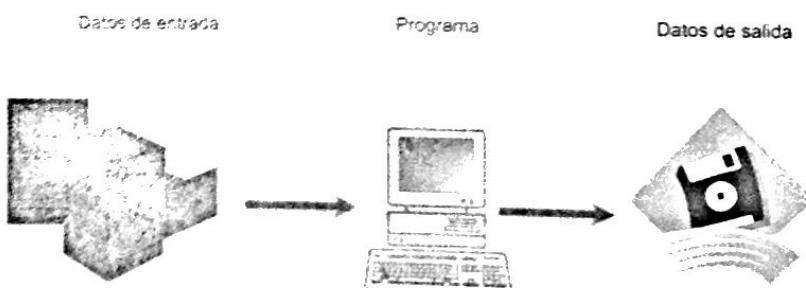
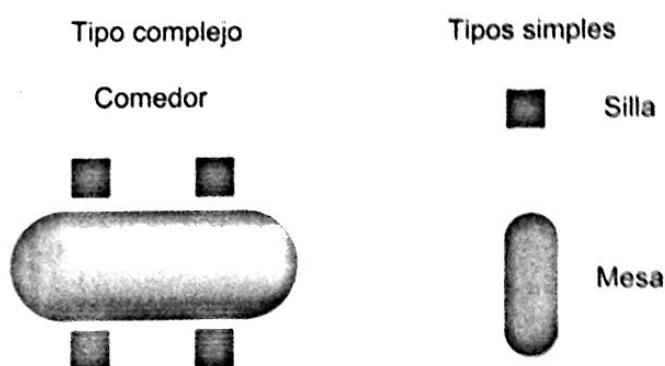


Figura 4.1. El programa como manipulador de datos

vez declarada una variable de un tipo determinado, a ésta se le asigna un trozo de memoria fijo, y este trozo no se podrá aumentar ni disminuir. Por ejemplo, un entero en lenguaje C ocupa 4 bytes de memoria y ese tamaño es el que puede representar. Normalmente, casi todos los tipos de datos son estáticos; la excepción son los punteros, que son dinámicos.

Los tipos de datos dinámicos son los denominados punteros (se verán más adelante). Un puntero no define un tipo de datos determinado, sino un apuntador en memoria a otro tipo de datos situados en memoria. Por ejemplo, un puntero que contenga el valor 3658 indica que apunta a la posición de memoria 3658, donde puede haber cualquier otro tipo de datos, comenzar un vector, etc. Estos tipos permiten tener un mayor control sobre la gestión de memoria en tus programas, manejar el tamaño de las variables en tiempo de ejecución y crear enlaces a variables cuyo contenido se desconoce en el momento del arranque del programa. Son el tipo de datos más complejo de entender y usar y muchos lenguajes de programación, como JAVA, no los incluyen. También son una de las fuentes de errores más importantes en los programas.

Atendiendo al número de elementos que pueden representar, se distingue entre tipos simples y complejos. Los tipos simples sólo representan un dato y son los tipos básicos en lenguaje C. Los tipos simples más básicos en C son: entero, carácter y real. La mayoría de los lenguajes de programación incluyen tipos básicos muy similares. Los tipos complejos, también denominados estructurados, se refieren a un conjunto o colección de elementos que pueden ser muy variados, pero que siempre están constituidos por tipos simples o por otros tipos complejos definidos previamente. Ejemplos de estos tipos complejos son los vectores, conjuntos o las estructuras de datos. La Figura 4.2 muestra un conjunto de comedor como un tipo complejo, así como los tipos simples que lo constituyen, mesa y silla.



Los tipos de datos son la forma de identificar los distintos datos que se pueden tratar en un programa. Aquellos tipos de datos no definidos, no existen para el lenguaje.

Figura 4.2. Tipos simples y complejos

Según la rigidez existente al manejar los tipos de datos, los lenguajes de programación se pueden clasificar como:

- Fuertemente tipados. En estos lenguajes todos los datos deben tener un tipo declarado explícitamente y existen restricciones en las operaciones y asignaciones en cuanto a los tipos de datos que en ellas intervienen. Normalmente, no permiten convertir un tipo de datos a otro de forma implícita. Un ejemplo de lenguaje de este tipo es JAVA.

- Débilmente tipados. En estos lenguajes el tipo de un dato no suele representar una gran restricción en las operaciones y asignaciones, pudiéndose intercambiar e interoperar datos de distinto tipo con gran facilidad. Normalmente, permiten convertir un tipo de datos a otro de forma implícita. C es un lenguaje con tipado débil.

Una vez definidas algunas generalidades de tipos de datos, vamos a ver cómo se representan los distintos tipos de datos en C y cómo se definen dentro de los programas.

## 4.2 Identificadores

Un *identificador* es un nombre que se asigna a los distintos elementos de un programa, como pueden ser variables, nombres de funciones, etc. Para construir identificadores en C se pueden usar caracteres alfabéticos en mayúsculas y minúsculas, así como los diez dígitos decimales (del 0 al 9) y el subrayado (\_), aunque no pueden empezar por un número. Por ejemplo, `as_DE_23` es un identificador válido, pero `as%DE` no lo es. Los distintos identificadores de una misma sentencia se separan mediante espacios en blanco o tabuladores. Además, es importante saber que C es sensible a mayúsculas y minúsculas, por lo que el identificador `pepe` es distinto al identificador `Pepe`. Por tanto, los siguientes identificadores son válidos en C:

```
casa    micasa1   _MES  
MES_1   mes_1     a890  
_890    _a890     M8a90
```

Como ejemplo de identificadores no válidos en C se muestran los siguientes:

- numero utiliza el carácter -
- 3numero utiliza un dígito como carácter inicial
- numero\_“ utiliza un carácter ilegal ”
- numero\$1 utiliza un carácter ilegal \$

### Ejercicio resuelto 4.1

Indique cuál de los identificadores siguientes es válido y cuál no en lenguaje C.

```
casa    mi-casa   mi*casa   micasa1  
_MES   MES_1     MES%UNO  mes$1  
a980   890a      _890      $a890
```

SOLUCIÓN.

Los identificadores `mi-casa`, `mi*casa`, `MES%UNO`, `mes$1` y `$a890` son inválidos porque usan separadores y símbolos no aceptados por la sintaxis del lenguaje. El identificador `890a` es inválido porque los identificadores no pueden comenzar por un número.

## Ejercicio resuelto 4.2

Indique si los siguientes identificadores son el mismo para el lenguaje C.

Traca, traca, traCa.

### SOLUCIÓN.

Son distintos porque C es sensible a los caracteres en minúscula y en mayúscula. Por tanto, los identificadores Traca, traca y traCa representan cosas distintas en lenguaje C.

Las *palabras reservadas* son identificadores que tienen un significado especial para el compilador del lenguaje de programación. C, al igual que todos los lenguajes de programación, define una serie de palabras reservadas, que se muestran en la Tabla 4.1. Así, por ejemplo, la palabra reservada `while` se utiliza para construir un determinado tipo de bucle y la palabra reservada `int` se emplea para definir tipos de datos enteros. Obviamente, las palabras reservadas no se pueden usar como identificadores de elementos del lenguaje definidos por el usuario (constantes, variables, etc.) Todas las palabras reservadas en C deben escribirse en minúsculas.



¡Cuidado! El lenguaje C es sensible a mayúsculas y minúsculas. Por ello, `casa` y `Casa` son identificadores distintos.

Tabla 4.1. Palabras reservadas en C

auto	else	long	typedef
break	enum	register	union
case	extern	return	unsigned
char	float	short	void
const	for	signed	volatile
continue	goto	sizeof	while
default	if	static	_Bool
do	inline	struct	_Complex
double	int	switch	_Imaginary

La forma de escribir los identificadores no está reglada en C, por lo que se pueden escribir como se deseé. Sin embargo, hay dos notaciones muy extendidas:

- Escribir las distintas palabras juntas y con la primera letra en mayúscula. Por ejemplo: `diasDelMes`.
- Escribir las distintas palabras separadas por el carácter subrayado (`_`). Por ejemplo: `dias_del_mes`.



Todas las palabras reservadas del lenguaje C, como las de casi todos los lenguajes de programación de computadores, están en inglés. Esto se debe a que hasta el momento prácticamente todos los lenguajes de programación se han creado en EEUU.

Se puede usar cualquiera de las dos, pero es importante seguir una convención para facilitar la legibilidad y la comprensión de los programas.

## 4.3 Variables y constantes

En esta sección se muestran las generalidades de variables y constantes en C. Los detalles específicos de estos elementos para cada tipo de datos se muestran en las secciones dedicadas a dichos tipos de datos.

### 4.3.1 Variables

Una *variable* es una representación alfanumérica de una posición de memoria. Como tal, se caracteriza por tres propiedades: posición de memoria que almacena el valor, tipo de datos almacenado y nombre que se refiere a esa posición de memoria. Una variable debe estar obligatoriamente ligada a un tipo de datos. Por ejemplo, `int a = 3;`, declara una variable `a` de tipo `int` cuyo valor inicial es 3. El tamaño de la zona de memoria, en bytes, dependerá del tipo de datos que se almacene en la variable. Las variables pueden contener diferentes valores durante la ejecución de un programa. La sintaxis de declaración de una variable es:

```
<tipo de datos> <nombre1> [<nombre2> .. <nombreN>]
```

Las *variables* son identificadores y, como todo identificador en C, deben escribirse siguiendo las reglas vistas anteriormente para construir los identificadores. Todas las variables en C deben definirse antes de su uso. Además, en una misma línea se pueden declarar varias variables del mismo tipo.



Una variable se denomina así porque el contenido puede «variarse» a lo largo del programa.

#### Ejercicio resuelto 4.3

Definición de variables de tipos elementales en C.

SOLUCIÓN.

```
int temperaturaHorno; //variable de tipo int
long numeroTelefono; //variable de tipo long
short k; //variable de tipo short
float interes; //variable de tipo float
```

El tipo de datos puede ser cualquiera de los tipos básicos de C, que estudiaremos en este capítulo, de los tipos complejos o de cualquier tipo declarado por el programador.

#### Ejercicio resuelto 4.4

Escriba un programa C que declare, usando nombres descriptivos, variables de C que representen meses del año, segundos en un minuto, una posición de la memoria de un computador y un valor lógico. Asigneles un valor y escríbalos por pantalla.

```
// Ejemplos de usos de variables
#define TRUE 1

int main(void)
{
    // Declaración de variables
    int mesDelAnyo;
    int segundos;
    int tamanyoMemoria;
    int valorLogico;
    // Parte ejecutiva del programa
    mesDelAnyo = 4;
    printf ("\n mesDelAnyo = %d", mesDelAnyo);
```

```

segundos = 34;
printf ("\n segundos = %d", segundos);
tamanyoMemoria = 20;
printf ("\n tamanyoMemoria = %d", tamanyoMemoria);
valorLogico = TRUE;
printf ("\n valorLogico = %i", valorLogico);
return (0);
}

```

En C se pueden definir variables sin *valor inicial* y variables con *valor inicial*. El formato de definición presentado anteriormente permite definir variables sin valor inicial. Cuando se desea asignar un valor inicial a una variable se utiliza el siguiente formato genérico:

<tipo\_de\_datos> <nombre1> = <valor>

### Ejercicio resuelto 4.5

Definición de variables con valor inicial en C.

SOLUCIÓN.

```

int temperaturaHorno = 136;      // variable de tipo int
long numeroTelefono= 916216574; // variable de tipo long
short k = 21;                  // variable de tipo short
float interes = 4.5;           // variable de tipo float

```

Esta práctica es muy buena, porque de esta forma se sabe a ciencia cierta el valor inicial de la variable, cuestión de la que no se puede estar seguro en caso contrario y que puede dar lugar a errores si se trabaja con las variables sin darles un valor (este asunto se estudiará más adelante).

### 4.3.2 Constantes

El lenguaje C permite definir constantes simbólicas que representan un valor determinado, que no cambia a lo largo del programa. Se definen mediante macros a través de la directiva `#define` usando la sintaxis siguiente:

`#define <nombre_constante> <valor>`

donde `nombre_constante` representa el nombre simbólico que se da a la constante y `valor` su valor. La palabra reservada `define` indica que la constante tiene un valor que se fija durante todo el período de vida que dura la ejecución de un programa y que el preprocesador debe sustituir las ocurrencias de `nombre_constante` por su valor. Por ejemplo, `#define PI 3.141516` define una constante PI que representa al número 3.141516.

Aunque no hay convenciones definidas respecto a los nombres de las constantes, se suelen escribir con letras mayúsculas, como se puede ver en el ejemplo siguiente.



Es interesante observar que no hay un tipo boolean como en JAVA. Se implementa mediante números enteros.



Nunca declare una variable sin valor inicial. Si lo hace, su valor será indefinido hasta que le asigne uno en el programa. En lenguaje C, este valor indefinido depende de cada compilador del lenguaje. En programas críticos en seguridad, dejar una variable sin valor inicial está prohibido por las normas de programación.



Una constante se denomina así porque su contenido no puede «variar» a lo largo del programa. Ahora bien, el lenguaje C no asegura que usted no pueda cambiar el valor de la constante. Se declaran así por compresión lógica del programa. Otros lenguajes, como ADA, no permiten cambiar una constante.

### Ejercicio resuelto 4.6

Definición de constantes de tipos elementales en C.

SOLUCIÓN.

```
#define TEMP_FIEBRE 37          // constante de tipo int
#define TEL_URGENCIAS 112        // constante de tipo short
#define PI 3.1416                // constante de tipo float
#define MENSAJE "Pulse Intro"    // constante de tipo String
#define VOCALPRIMERA 'a'         // constante de tipo char
#define VERDAD true              // constante de tipo boolean
#define VELOCIDAD_LUZ 300000     // constante de tipo int
#define PELIGRO "Peligro. No pasar" // constante de tipo String
#define PI_DOUBLE 3.1416          // constante de tipo double
```

Observe que las sentencias que definen macros no tienen punto y coma al final de las mismas.

### Ejercicio resuelto 4.7

Escriba un programa que declare, usando nombres descriptivos, constantes en C que representen los meses del año, los segundos de un minuto, que el tamaño de la memoria de un computador es 2 Kbytes, un valor lógico verdadero y la razón entre grados Fahrenheit y Celsius. El programa debe imprimir los valores por pantalla.

SOLUCIÓN.

```
// Ejemplos de uso de constantes
// Declaración de constantes
#define MESESDELANYO 12          // Meses del año
#define SEGUNDOS_MINUTO 60        // Segundos en un minuto
#define TAMANYOMEMORIA 2048      // Tamaño de memoria
#define VERDADERO 1               // Constante verdadero
#define CELSIUSFARENHEIT 1.8      // Conversión de grados
#define PRECIO_DINERO 3.9         // Tipo de interés

int main(void)
{
    // Parte ejecutiva del programa
    printf ("\n MESESDELANYO = %d", MESESDELANYO);
    printf ("\n SEGUNDOS_MINUTO = %d", SEGUNDOS_MINUTO);
    printf ("\n TAMANYOMEMORIA = %d", TAMANYOMEMORIA);
    printf ("\n VERDADERO = %d", VERDADERO);
    printf ("\n CELSIUSFARENHEIT = %f", CELSIUSFARENHEIT);
    printf ("\n Tipo de interés = %f", PRECIO_DINERO);
    return(0);
}
```

Observe que no es posible asignar valor a las constantes después de su declaración, porque ya lo tienen y no se puede modificar. Sin embargo, nunca se debe confundir una constante con una variable con valor inicial. Hay dos diferencias fundamentales:

1. Su declaración se diferencia porque las constantes tienen el calificador **#define**.
2. El valor de la constante no puede cambiar, por lo que se obtiene un error si se trata de asignar un valor a una constante.

#### Ejercicio propuesto 4.1

Escriba un programa que declare, usando nombres descriptivos, una constante en C que represente el número de meses del año y una variable para almacenar un número de mes, iniciada con el valor 5. El programa debe imprimir por la pantalla el orden del mes dentro del año y la porción de año transcurrido.

#### Ejercicio resuelto 4.8

Escriba un programa que declare la constante PI como 3.1416, que calcule el área de un círculo, cuyo radio se pide por pantalla, y la imprima por pantalla.

SOLUCIÓN.

```
// Ejemplos de usos de constantes en cálculos
#include <stdio.h>

int main(void)
{
    // Declaración de constantes y variables
    #define PI 3.1416 // constante PI
    float area = 0;
    int radio = 5; // radio del círculo

    // Parte ejecutiva del programa
    // lectura del radio desde la consola
    printf ("\n Radio del circulo: ");
    scanf ("%d", &radio);

    // Área de un círculo
    area = PI * radio * radio;
    printf ("\n El área de un círculo de radio = %d es
            %f ", radio, area);
    return(0);
}
```

Terminaremos esta sección indicando que los distintos tipos de constantes en C son:

- Caracteres: 'a', 'b'.
- Valores enteros, que pueden expresarse de distintas formas:
  - notación decimal: 987;
  - notación hexadecimal: 0x25 o 0X25;
  - notación octal: 034;

- enteros sin signo: 485U;
  - enteros de tipo long: 485L;
  - enteros sin signo de tipo long: 485UL;
  - valores negativos (signo menos): -987.
- Valores reales (coma flotante):
- ejemplos: 12, 14, 8., .34;
  - notación exponencial: .2e+9, 1.04E-12;
  - valores negativos (signo menos): -12 -2e+9.

## 4.4 Tipos de datos elementales

En C hay ocho tipos de datos elementales, ya sean básicos o fundamentales, a partir de los cuales se construyen todos los demás. Estos tipos, que se muestran en la Tabla 4.2, sirven principalmente para hacer operaciones aritméticas, representación de caracteres y de valores lógicos.

Los tipos elementales se denominan así porque están disponibles en cualquier implementación de un compilador del lenguaje C, ya que han sido definidos en el estándar del lenguaje.

En esta sección se estudian los tipos elementales y sus características. Además, se aprende a usar dichas características por medio de ejemplos.

Tabla 4.2. Tipos elementales de C

Tipo	Significado	Tamaño en bytes
char	carácter	1
int	entero	2 ~ 4
short	entero corto	2
long	entero largo	4
long long	entero largo	8
unsigned char	carácter sin signo	1
unsigned	entero sin signo	2 ~ 4
unsigned char	entero corto sin signo	2
unsigned long	entero largo sin signo	4
unsigned long long	entero largo sin signo	8
float	coma flotante, real	4
double	coma flotante largo	8
long double	coma flotante de doble precisión extendida	16

### 4.4.1 Operadores

En el lenguaje C se definen operaciones básicas sobre los tipos elementales. Por ejemplo, los operadores aritméticos permiten realizar operaciones matemáticas básicas con números enteros. Una expresión como la siguiente:

```

numero = bitOcteto + (5 * 4);
resultado = numero * 5 / 4 + 3;

```

es una expresión aritmética compuesta por operadores aritméticos y operandos, donde los operandos deben ser de tipo entero. La Tabla 4.3 muestra los operadores que se pueden aplicar a cada tipo, ordenados de mayor a menor precedencia. El orden de precedencia indica qué operador se aplica en primer lugar cuando hay varios presentes en una expresión aritmética. Así, en el ejemplo anterior, resultado se calcula en el orden siguiente:

1. resultado = numero \* 5 / 4
2. resultado = resultado + 3;

La mejor forma de evitar problemas por la precedencia de operadores es usar paréntesis para indicar cómo se deben ejecutar las operaciones. Por ejemplo:

```
resultado = (numero * (5 / 4)) + 3;
```

Tabla 4.3. Precedencia de operadores en C

Operador	Significado	Tipos aceptados
x++, x--	Postincremento/Postdecremento	Alfanuméricos
++x, --x, +x, -x	Postincremento/Postdecremento	Alfanuméricos
!	No lógico	Booleanos
(tipo)expr	Creación o conversión de tipo	Todos
*, /, %	Multiplicación, división, módulo	Alfanuméricos
+, -	Suma, resta	Alfanuméricos
<<, >>, >>>	Desplazamiento binario	Todos
<, <=, >, >=	Comparaciones de orden	Alfanuméricos
==, !=	Test de igualdad	Todos
&	AND binario	Todos
	OR binario	Todos
&&	AND lógico	Booleanos
	OR lógico	Booleanos
=, +=, -=, *=, /=, %=, &=	Asignación	Todos (del mismo tipo)
^=,  =, <<=, >>=, >>>=		

El operador `sizeof` devuelve el tamaño en bytes que ocupa un determinado tipo de datos o variable. El resultado de este operador podrá ser distinto dependiendo del computador en el que se use, ya que el número de bytes utilizado para representar cada uno de los datos básicos depende de la forma en la que los representa cada computador.

En las secciones siguientes se mostrarán ejemplos de uso de estos operadores con cada tipo de datos.

#### 4.4.2 Tipos numéricos enteros

Los tipos de datos que permiten representar números enteros en C son cuatro: `short`, `int`, `long` y `long long`. Todos ellos permiten representar números positivos



La precedencia de los operadores indica el orden en que se ejecutan las operaciones cuando hay varios operadores y no están separados por paréntesis. La precedencia es peligrosa porque puede dar resultados no deseados si no se conocen exactamente las reglas del lenguaje. Las reglas de precedencia son básicas para sumas, restas, multiplicaciones y divisiones. Pero, ¿qué ocurre con operaciones a nivel de bit, lógicas, potencias, etc.? Lo mejor es usar paréntesis para delimitar bien el alcance de una operación.

y negativos sin partes decimales. Por ello, los enteros son el tipo de datos que se deben usar para todo aquello que se pueda contar en unidades enteras.

### Ejercicio resuelto 4.9

Defina variables para representar las personas que hay en un teatro, las líneas telefónicas de un país, los días de la semana o la posición de un bit en octeto.

SOLUCIÓN.

```
int espectadores = 0;  
long lineasTelefonicas = 0;  
short diaSemana = 7;  
short bitOcteto = 0;
```

A pesar de que existe un valor inicial por defecto, siempre que se conozca el valor inicial de que parten los valores es conveniente asignar ese valor a la variable en el momento de su declaración. En caso de que no se conozca, o no se haya definido ese valor, conviene dar igualmente un valor inicial.

Además, los tipos de datos enteros se pueden especificar de diferentes formas. A continuación se muestra una lista en la que se presentan las diferentes formas de especificar cada uno de los tipos de datos enteros:

- short, signed short, short int o signed short int;
- unsigned short o unsigned short int;
- int, signed o signed int;
- unsigned o unsigned int;
- long, signed long, long int o signed long int;
- unsigned long o unsigned long int;
- long long, signed long long, long long int o signed long long int;
- unsigned long long o unsigned long long int.



Es importante elegir el tipo adecuado, pero si tiene dudas elija el de más amplio rango. Imagine que elige un **short** y quiere asignarle el valor 1245678. No se puede representar con los 16 bits del tipo **short**, que admite hasta 64000, por lo que se truncará y quedará un valor que no se parecerá en nada al original.

El truncado se puede hacer a derechas o izquierdas dependiendo del compilador. Lo normal es que se pierda la parte «alta» del número. Es decir, la de más valor.

La diferencia principal entre estos tipos está en su capacidad de representación y en si permiten representar signo o no. La capacidad de representación viene dada por el espacio de memoria que ocupa cada tipo. La Figura 4.3 muestra una comparativa de los bytes que ocupan cuatro tipos básicos. La palabra reservada **unsigned** indica que se trata de un número sin signo, es decir, positivo.

La Tabla 4.4 muestra los rangos de valores de cada tipo y su valor inicial por defecto. El rango de un tipo se puede obtener mediante los calificadores **MIN** y **MAX** de cada tipo respectivamente, que se pueden encontrar en el archivo **limits.h** del sistema. Este archivo incluye, entre otros, los valores mínimo y máximo que se pueden representar para cada uno de los tipos de datos en el computador en el que esté ejecutando sus programas.

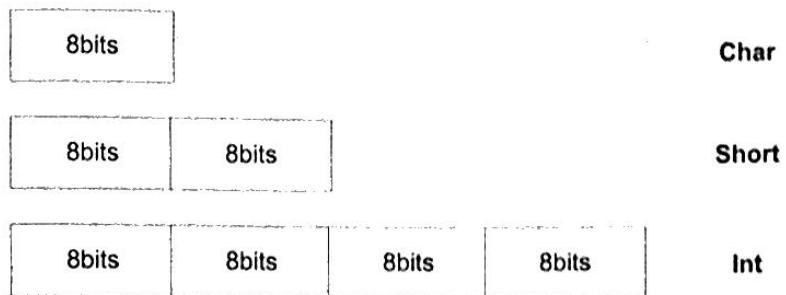


Figura 4.3. Tamaño en memoria de tipos de datos enteros

Tabla 4.4. Rangos de los tipos de datos para enteros

Tipo	Longitud en bits	Valor por defecto	Valor mínimo	Valor máximo
short	16	0	-32768	+32767
int	32	0	-2147483648	+2147483647
long	64	0	-9223372036854775808	+9223372036854775807

### Ejercicio resuelto 4.10

Programa de ejemplo que muestra los tamaños utilizados en el computador en el que se ejecuta el programa para los tipos de datos enteros.

SOLUCIÓN.

```

/* Programa que muestra los tamaños utilizados
 * en el computador en el que se ejecutó
 * el programa para los tipos de datos enteros
 * (archivo limits.h).
 */
#include <limits.h>

int main(void)
{
    printf("\n número de bits por char: %d", CHAR_BIT);
    printf("\n mínimo valor para un signed char: %d", SCHAR_MIN);
    printf("\n máximo valor para un signed char %d", SCHAR_MAX);
    printf("\n máximo valor para un unsigned char, el mñ. será 0 %d",
           UCHAR_MAX);

    printf("\n mínimo valor para un signed short %d", SHRT_MIN);
    printf("\n máximo valor para un signed short %d", SHRT_MAX);
    printf("\n máximo valor para un unsigned short %d", USHRT_MAX);
    printf("\n mínimo valor para un int %d", INT_MIN);
    printf("\n máximo valor para un int %d", INT_MAX);
    printf("\n máximo valor para un unsigned int %d", UINT_MAX);
    printf("\n mínimo valor para un long %ld", LONG_MIN);
    printf("\n máximo valor para un long %ld", LONG_MAX);
    printf("\n máximo valor para un unsigned long %ld", ULONG_MAX);

    return(0);
}

```

Si se asigna a una variable de un tipo un valor que está fuera de su rango de representación, como por ejemplo:

```
unsigned int = -1;
```

no se obtiene un error de compilación, ni ningún otro aviso. Éste es uno de los principales problemas de C: su permisividad con los tipos. Por ello, hay que ser cuidadoso y comprobar que los valores de entrada están dentro de sus rangos.

Una forma más sencilla de comprobar el tamaño que ocupa cada tipo en memoria es usar el operador `sizeof`.

### Ejercicio resuelto 4.11

Ejemplo que muestra, usando `sizeof`, los tamaños en bytes que ocupan los tipos enteros en memoria en el computador.

SOLUCIÓN.

```
/* Este programa muestra para cada uno de los tipos de
 * datos básicos enteros de C el tamaño que ocupa en
 * bytes. La salida de este programa será distinta
 * dependiendo del computador en el que lo ejecute, ya
 * que el número de bytes utilizado para representar
 * cada uno de los datos básicos depende de la forma
 * en la que los representa cada computador.
 */
#include <stdio.h>
int main(void)
{
    printf("Un char ocupa %d bytes\n", sizeof(char));
    printf("Un signed char ocupa %d bytes\n", sizeof(signed char));
    printf("Un unsigned char ocupa %d bytes\n",
           sizeof(unsigned char));
    printf("Un int ocupa %d bytes\n", sizeof(int));
    printf("Un signed int ocupa %d bytes\n", sizeof(signed int));
    printf("Un unsigned int ocupa %d bytes\n", sizeof(unsigned int));
    printf("Un short ocupa %d bytes\n", sizeof(short));
    printf("Un signed short ocupa %d bytes\n", sizeof(signed short));
    printf("Un unsigned short ocupa %d bytes\n",
           sizeof(unsigned short));
    printf("Un long ocupa %d bytes\n", sizeof(long));
    printf("Un signed long ocupa %d bytes\n", sizeof(signed long));
    printf("Un unsigned long ocupa %d bytes\n",
           sizeof(unsigned long));
    printf("Un long long ocupa %d bytes\n", sizeof(long long));
    printf("Un signed long long ocupa %d bytes\n",
           sizeof(signed long long));
    printf("Un unsigned long long ocupa %d bytes\n",
           sizeof(unsigned long long));
    printf("Un float ocupa %d bytes\n", sizeof(float));
    printf("Un double ocupa %d bytes\n", sizeof(double));
    printf("Un long double ocupa %d bytes\n", sizeof(long double));
    return(0);
}
```

## ► Operaciones con enteros

Una vez vistos los tipos, sus rangos y los operadores básicos, se presentan algunos ejemplos de operaciones con enteros.

### Ejercicio resuelto 4.12

Escriba un programa que imprima los cinco primeros números enteros positivos, calcule su suma, su multiplicación y haga la división de la multiplicación por 3.

SOLUCIÓN.

```
// Operaciones con números enteros
int main(void)
{
    // Declaración de variables
#define DIVISOR 3
    int i = 1; // Variable para positivos
                // Variables para suma, mult. y división
    int suma = 0, mult = 1, divis =1;

    // Parte ejecutiva del programa
    suma = i;
    mult = i;
    printf ("\n\n Número = %d", i++);
    suma = suma + i;
    mult = mult * i;
    printf ("\n Número = %d", i++);
    suma = suma + i;
    mult = mult * i;
    printf ("\n Número = %d", i++);
    suma = suma + i;
    mult = mult * i;
    printf ("\n Número = %d", i++);
    suma = suma + i;
    mult = mult * i;
    printf ("\n Número = %d", i);
    printf ("\n Suma = %d", suma);
    printf ("\n Multiplicación = %d", mult);
    divis = mult / DIVISOR;
    printf ("\n División = %d", mult / DIVISOR);
    return(0);
}
```

### Ejercicio resuelto 4.13

Escriba un programa que lea un número por la entrada estándar, lo multiplique por 20 e imprima su división por 10. A continuación debe sumar dicho número a la multiplicación y volver a imprimir su división por 10. Si el resto no es cero, debe imprimirlo también.

SOLUCIÓN.

```
// División entera y operador módulo
#include <stdio.h>
```

```

int main(void)
{
    // Declaración de variables
#define DIVISOR 10
    int i = 5; // Variable para positivos
    int mult = 1, divis =1; // Variable para suma,
                           //mult. y división

    // Parte ejecutiva del programa
    // lectura del número desde la consola
    printf ("\n Escriba un numero entero: ");
    scanf ("%d", &i);
    // operaciones solicitadas
    mult = i * 20;
    printf ("\n %d * 20 / 10 es %d ", i, (mult / DIVISOR));
    mult = mult + i;
    printf ("\n División entera:
            (%d * 20 + %d) / 10 es %d ", i, i, (mult / DIVISOR));
    if ( (mult % DIVISOR) != 0 )
        printf ("\n Resto: (%d * 20 + %d)
                módulo 10 es %d ", i, i, (mult % DIVISOR));
    return(0);
}

```



Observe que la división entera trunca el resultado al entero más próximo por debajo. Por ejemplo, el valor de  $23/4$  es 5.

#### 4.4.3 Tipos numéricos reales

En cuanto a los números reales o números en coma flotante, existen tres tipos básicos en lenguaje C, que se muestran en la Tabla 4.5 junto con su tamaño en bytes. La diferencia entre ellos se encuentra en la precisión de los números que se pueden representar. Es decir, el conjunto de valores de tipo **float** es un subconjunto del conjunto de valores de tipo **double** y el conjunto de valores de tipo **double** es un subconjunto del conjunto de valores de tipo **long double**.

Ambos se pueden representar mediante tres tipos de notación:

- Base decimal utilizando un punto para separar la parte entera de la decimal. Las siguientes constantes representan valores reales utilizando notación decimal-punto:

1234.898 0.233.

Tabla 4.5. Tipos de datos básicos reales en C

Tipo	Tamaño	Representa
float	4 bytes	Nº en coma flotante
double	8 bytes	Nº en coma flotante de doble precisión
long double	16 bytes	Nº en coma flotante de doble precisión extendida

- Mantisa decimal con base decimal y exponente entero. En este caso la base se sustituye por las letras E o e. Las siguientes constantes representan valores reales utilizando notación exponencial decimal:

`23.4e2` representa  $23.4 \times 10^2$ ;  
`-1.9E-18` representa  $-1.9 \times 10^{-18}$ ;  
`.00988e4` representa  $0.00988 \times 10^4$ .

- Mantisa hexadecimal con base dos y exponente entero. En este caso la base se sustituye por las letras P o p.

Las siguientes constantes representan valores reales utilizando notación exponencial decimal:

`0x23.4p14` representa  $0x23.4 \times 2^{14}$ ;  
`0x654.4p2` representa  $0x654.4 \times 2^2$ .

La precisión de las constantes en coma flotante se puede especificar con las letras F y L (en mayúscula o en minúscula). La primera se utiliza para indicar valores de precisión normal y la segunda para valores de doble precisión. Así la constante `2.34E48L` representa un valor en coma flotante de doble precisión, mientras que `2.34E48F` representa el mismo valor, pero con precisión normal.

## ► Operaciones con reales

Una vez vistos los tipos, sus rangos y los operadores básicos, se presentan algunos ejemplos de operaciones con reales.

### Ejercicio resuelto 4.14

Escriba un programa que lea un número real por la entrada estándar, lo multiplique por 20.0 e imprima su división por 10. A continuación debe sumar el número a la multiplicación y volver a imprimir su división por 10. Si el resto no es cero, debe imprimirla también.

SOLUCIÓN.

```
// División de números reales

#include <stdio.h>

int main(void)
{
    // Declaración de variables
#define DIVISOR 1.0E+1F
    int i = 5; // Variable para positivos
    float mult=1, divis=1; // Variable para suma, mult.
                           // y división

    // Parte ejecutiva del programa
    // lectura del número desde la consola
    printf ("\n Escriba un numero entero: ");
    scanf ("%d", &i);
```



Observe que la división real no trunca el resultado al entero más próximo por debajo. Por ejemplo,  $23/4$  da  $5.75$ .

```
// operaciones solicitadas
mult = i * 20.0;
printf ("\n %d * 20.0 / 10.0 es %f ", i, (mult / DIVISOR));
mult = mult + i;
printf ("\n División real:
        (%d * 20.0 + %d) / 10.0 es %f ", i, i, (mult / DIVISOR));
if ( ((int)mult % (int)DIVISOR) != 0)
    printf ("\n Resto: (%d * 20.0 + %d)
            módulo 10.0 es %f ", i, i, ((int)mult % (int)DIVISOR));
return(0);
}
```

### Ejercicio resuelto 4.15

Escriba un programa que calcule la ecuación del espacio recorrido por un objeto que se mueve a velocidad constante. Había recorrido un espacio inicial de 5.5 m y se mueve a velocidad constante de 3.2 m/s. Debe pedir por la entrada estándar los segundos que se mueve.

SOLUCIÓN.

```
// Cálculo del espacio que recorre un móvil
#include <stdio.h>
int main(void)
{
    // Declaración de constantes y variables
    #define EspacioInicial 5.5F // Espacio inicial
    #define Velocidad 3.2F      // Velocidad
    float tiempo = 22.3F;       // Tiempo del recorrido
    float espacio = 0;

    // Parte ejecutiva del programa
    // lectura del tiempo desde la consola
    printf ("\n Tiempo de desplazamiento: ");
    scanf ("%f", &tiempo);
    // operaciones solicitadas
    espacio = EspacioInicial + Velocidad * tiempo;

    // Salida de resultados
    printf ("\n Espacio recorrido = %f metros", espacio );
    return(0);
}
```

#### ► La biblioteca math

La biblioteca `math` proporciona funciones matemáticas que permiten realizar cálculos complejos de tipo trigonométrico, con números aleatorios, etc. Se puede obtener una descripción completa de las funciones que proporciona la biblioteca `math` en la ayuda de su entorno de desarrollo (por ejemplo, `man sqrt`). A continuación se muestran algunos ejemplos de funciones de esta clase:

$x = \cos(a)$  devuelve el coseno de a;  
 $x = \text{atan}(a)$  devuelve el arco tangente de a;  
 $x = \exp(a)$  devuelve el valor de  $e$  (2.718) elevado a a;  
 $x = \text{pow}(a, b)$  devuelve el valor de a elevado a b.

### Ejercicio resuelto 4.16

Escriba un programa que calcule las raíces de la ecuación cuadrática:  $a*x^2 + b*x + c = 0$  y las imprima por pantalla. Debe pedir por pantalla los coeficientes a, b y c.

SOLUCIÓN.

Para resolver esa ecuación, hay que calcular:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```

// Cálculo de las raíces de una ecuación cuadrática
#include <stdio.h>
#include <math.h>

int main (void)
{
    // Declaración de constantes y variables
    float a = 2.0;
    float b = -5.0;
    float c = 2.0;
    float raiz1 = 0.0, raiz2 = 0.0;
    float discriminante = 0.0;

    // Parte ejecutiva del programa
    // lectura del número desde la consola
    printf ("\n Escriba el coeficiente a: ");
    scanf ("%f", &a);
    printf ("\n Escriba el coeficiente b: ");
    scanf ("%f", &b);
    printf ("\n Escriba el coeficiente c: ");
    scanf ("%f", &c);

    // Cálculo de las raíces e impresión de resultados
    printf ("\n Las raíces de %f x^2 + %f x + %f ", a, b, c );
    discriminante = (float)(pow(b,2) - 4*a*c);
    if (discriminante < 0)
        printf (" No existen ");
    else
    {
        raiz1 = (float)((-b + sqrt(discriminante)) / 2*a);
        raiz2 = (float)((-b - sqrt(discriminante)) / 2*a);
        printf (" Son %f y %f\n", raiz1, raiz2 );
    }
    return(0);
}

```

#### Ejercicio propuesto 4.2

Escriba un programa que permita calcular la cantidad mensual que hay que pagar y el pago total para un préstamo de una cantidad, un interés anual, y una duración en años dada. La fórmula que hay que usar para calcular el pago mensual es la siguiente:

$$\text{cuota} = \frac{\text{cantidad} * \text{interesMensual}}{1 - \left( \frac{1}{(1 + \text{interesMensual})^{\text{numeroPagos}}} \right)}$$

Particularice el ejemplo para 100000 euros, tipo de interés del 3,5% y una duración de 15 años.

#### 4.4.4 Números complejos

C99 introduce nuevos tipos de datos para trabajar con números complejos. Estos tipos son:

```
float _Complex  
double _Complex  
long double _Complex
```

Estos tipos de datos permiten representar números complejos de la forma  $u + iv$ , donde  $u$  y  $v$  se consideran números reales. Las constantes de este tipo se representan como  $a+Ib$ , donde  $a$  y  $b$  se representan como valores reales e  $I$  se utiliza para el número  $i$ .

También existen tipos de datos para trabajar con números imaginarios de la forma  $iu$ . Estos tipos son:

```
float _Imaginary  
double _Imaginary  
long double _Imaginary
```

Para poder trabajar con estos tipos de datos es necesario que incluya en sus programas el archivo `complex.h`.

#### 4.4.5 Tipo carácter

C proporciona un tipo básico para manipular caracteres. El tipo `char` permite representar valores consistentes en un único carácter, tales como:

'a', 'x', 'D', '&', '7'

Los caracteres admitidos como válidos en C son aquéllos incluidos en la tabla del estándar ASCII extendido, para poder representar los símbolos de lenguajes distintos al inglés. Actualmente, el estándar UNICODE incluye 256 caracteres, por lo que son necesarios 8 bits para representar un carácter en C. Habitualmente, la mayoría de los lenguajes de programación usan el estándar ASCII para representar los caracteres. El estándar ASCII incluye 128 caracteres, que representan el alfabeto inglés, los dígitos del 0 al 9 y algunos caracteres especiales. Posteriormente se amplió a 256



Aunque C99 introduce estos nuevos tipos de datos, sin embargo no es obligatorio que un compilador concreto los soporte. Por lo que puede encontrarse con compiladores en los que no se pueda trabajar con estos tipos de datos.

De hecho, es lo habitual.

caracteres para incluir los caracteres existentes en alfabetos europeos comunes (como la á o la Ä). Para ser compatible con todos estos lenguajes, y con la mayoría de las aplicaciones del mundo occidental, en C también se usan los 256 primeros caracteres para representar el código ASCII extendido. La Figura 4.4 muestra la tabla de caracteres ASCII.

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	:
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	DEL		

Figura 4.4. Tabla de caracteres ASCII

Las variables de tipo carácter se definen como cualquier otra en C:

```
primeravocal = 'a';
char letra;
char numero = '0';
unsigned char = 'á';
```

Observe que para representar caracteres por encima del ASCII 127, es decir pasar al ASCII extendido, es necesario declarar las variables como `unsigned char`. En caso contrario, el octavo bit del byte de un `char` se toma como bit de signo. En las transmisiones de datos en ASCII estándar (hasta 127 caracteres), el octavo bit se usa como comprobación de paridad para ver que el contenido del bit es correcto. Si se quieren transmitir caracteres en español, es decir, usando acentos, eñes, etc., es necesario usar la parte alta del ASCII extendido y declarar las variables como `unsigned char`.

#### Ejercicio propuesto 4.3

Escriba un programa que declare una variable `char` y otra `unsigned char`. Asigne la «ñ» a ambas e imprima por pantalla. ¿Qué ocurre?

Se puede mostrar el código ASCII de un carácter con sólo convertirlo a un tipo `int`. Igualmente, se puede obtener el carácter asociado a un número entero entre 0 y 127 con sólo convertirlo a `char`, como se muestra en las sentencias siguientes:

```
int codigo_a = (int)'a';
char primeraLetra = (char) 97;
```



El lenguaje C no entiende otras codificaciones de caracteres como UTF-1 o UNICODE. Sólo entiende ASCII o ASCII extendido hasta el código de carácter 256, pero éstos últimos ya deben representarse como `unsigned char`.

También se pueden obtener estos valores asignando caracteres a enteros y viceversa. Esta técnica es problemática porque da lugar a pérdidas de información y

redondeos no deseados. Tenga en cuenta que al asignar un entero a un carácter sólo se toma el primer byte.

### Ejercicio resuelto 4.17

Escriba un programa que obtenga la representación de 'a' en letra y en número y que pida un número entero, imprima su carácter asociado y luego lo imprima como un entero.  
SOLUCIÓN.

```
// Conversión a y desde caracteres.

#include <stdio.h>

int main (void)
{
    // Declaración de variables
    int codigo_a;
    char primeraLetra = 97;
    char caracter = 97;
    int numero;

    // Parte ejecutiva del programa
    // lectura del número desde la consola
    codigo_a = (int)'a';
    printf ("\n El código de 'a' es: %d ", codigo_a);
    printf ("\n La primera letra es: %c ", caracter);

    printf ("\n Escriba un número positivo: ");
    scanf ("%d", &numero);
    printf ("\n Su carácter correspondiente es: %c ", (char)numero);
    caracter = numero;
    printf ("\n Si se convierte el carácter asociado a número
se obtiene: %d", caracter);
    return(0);
}
```



Cuidado con la asignación de enteros a caracteres.  
Si introduce un número por encima de 127, como 234, observará que al imprimirla como número no imprime 127 sino -22. ¿Por qué?  
Pues porque se usa representación en complemento a 2 y con el octavo bit a 1 se entiende que es un número entero.

Hay ciertos caracteres que son problemáticos, porque se usan en el lenguaje C para otras cosas (como ", que delimita la cadena de formato) o que no se pueden representar directamente, como ocurre con los códigos ASCII entre 0 y 31 denominados caracteres de control (como por ejemplo el salto de línea). Estos caracteres se usan mediante secuencias de escape. Una *secuencia de escape* es un carácter que está representado por el carácter \ seguido por una letra o conjunto de dígitos. La Tabla 4.6 muestra las secuencias de escape definidas en C.

### Ejercicio resuelto 4.18

Escriba un programa que muestre por pantalla las letras minúsculas y su código ASCII.  
SOLUCIÓN.

La primera letra minúscula es la 'a'. Por ello, se comienza buscando su código como punto de partida.

Tabla 4.6. Secuencias de escape

Secuencia	Significado
\n	nueva línea
\t	tabulador
\b	backspace
\r	retorno de carro
\"	comillas
\'	apóstrofo
\\\	backslash
\?	signo de interrogación

```
// Programa que imprime las minúsculas
int main(void)
{
    // Declaración de constantes y variables
    #define primerCodigo 'a'
    int i;

    // Parte ejecutiva del programa
    printf ("Las letras minúsculas y sus códigos ASCII son: \n");
    for (i = primerCodigo; i < primerCodigo+26; i++)
        printf ("\n Carácter: %c Valor: %d", (char) i, i);
    return(0);
}
```

Las operaciones aplicables a los caracteres son todas aquellas aplicables a los tipos alfanuméricos. Por tanto, se puede operar con ellos o sus códigos, se pueden asignar y se pueden comparar.

### Ejercicio resuelto 4.19

Escriba un programa que ordene de menor a mayor dos caracteres que se leen por pantalla y los escriba ordenados por pantalla.

SOLUCIÓN.

```
// Programa que ordena caracteres
#include <stdio.h>

int main(void)
{
    // Declaración de constantes y variables
    char car1;
    char car2;

    // Parte ejecutiva del programa
    printf ("\n Escriba los dos caracteres: ");
    scanf ("%c", &car1);
    scanf ("%c", &car2);
```

```

// Ordenar los caracteres
printf ("\n Los caracteres ordenados son ");
if (car1 < car2)
    printf ("%c y %c", car1, car2 );
else
    printf ("%c y %c", car2, car1 );
return(0);
}

```

#### 4.4.6 Tipo lógico

El lenguaje C, a diferencia de otros muchos lenguajes de programación, no ha dispuesto tradicionalmente de un tipo de datos booleano para representar los valores verdadero (*true*) o falso (*false*). Las variables para representar valores lógicos son enteras, por tanto pueden tener cualquier valor que se quiera para representar el verdadero y el falso, aunque para suplir esta carencia se suelen usar los valores 1 y 0 respectivamente. Para que no haya problemas de criterio, se suelen definir las dos constantes siguientes:

- **#define VERDAD 1.** Indicar un valor lógico verdadero.
- **#define FALSO 0.** Indicar un valor lógico falso.

A las variables de tipo entero se les pueden aplicar los operadores que se muestran en la Tabla 4.3, de entre los cuales merece la pena destacar el significado de los operadores lógicos NOT, AND y OR, cuyos resultados se ajustan a tablas lógicas que se muestran en la Tabla 4.7.

Tabla 4.7. Tablas de operaciones lógicas

A	B	$\neg A$	$A \& \& B$	$A \mid\mid B$
0	0	1	0	0
1	0	0	0	1
0	1		0	1
1	1		1	1

A partir de las tablas anteriores se puede evaluar cualquier expresión lógica, aunque no esté expresada en variables lógicas. En el ejercicio siguiente se puede observar cómo se trabaja con operadores lógicos y cómo el resultado de un operador relacional es de tipo booleano.

#### Ejercicio resuelto 4.20

Escriba un programa que declare dos macros con valores *verdad* y *falso* y dos variables enteras con valores 5 y 4. Haga con ellas todas las operaciones lógicas posibles.

**SOLUCIÓN.**

```

// Programa que hace operaciones lógicas
int main(void)
{
    // Declaración de constantes y variables
#define verdad 1
#define falso 0
    int i = 5, j = 4;
    int aux;

    // Parte ejecutiva del programa
    //Operadores lógicos con booleanos
    printf("\n operaciones lógicas ...");
    printf("\n NOT verdad es %d ", !verdad);
    printf("\n NOT falso es %d ", !falso);
    aux = verdad && falso;
    printf("\n verdad AND falso es %d ", aux);
    aux = verdad || falso;
    printf("\n verdad OR falso es %d ", aux);

    //Operadores relacionales
    printf(" Mayor o igual que ..");
    printf("\n (i >= j) = %d ", (i >= j)); //verdad
    printf("\n (j >= i) = %d ", (j >= i)); //falso
    printf("\n (i != j) = %d ", (i != j)); //verdad

    //Operadores lógicos con enteros
    printf("\n operaciones lógicas con enteros...");
    printf("\n NOT(i >= j) = %d ", !(i >= j)); //falso
    aux = (i >= j) && (j >= i);
    printf("\n (i >= j) AND (j >= i) = %d ", aux); //falso
    aux = (i >= j) || (j >= i);
    printf("\n (i >= j) OR (j >= i) = %d ", aux); //verdad
    return(0);
}

```

**4.4.7 Tipos enumerados**

Un *tipo enumerado* permite definir un conjunto de constantes simbólicas con valor entero.

**Ejercicio resuelto 4.21**

Programa que define un tipo enumerado para los días de la semana, una variable del tipo e imprime los días de la semana.

**SOLUCIÓN.**

```

// Programa que hace operaciones con días de la semana
int main(void)

```



En la nueva revisión del estándar de C que apareció en diciembre de 1999, se ha introducido el tipo de datos `_Bool`, con los valores `true` y `false`, disponible en el archivo de cabecera `stdbool.h`. Sin embargo, todavía no está disponible en muchos compiladores de C.

```

{
    // Declaración de constantes y variables
    enum DiasSemana {lunes, martes, miercoles, jueves,
                      viernes, sabado, domingo};
    enum DiasSemana dia;
    dia = lunes;
    printf ("\n Primer día de la semana: %d", dia);
    dia = martes;
    printf ("\n Segundo día de la semana: %d", dia);
    dia = miercoles;
    printf ("\n Tercer día de la semana: %d", dia);
    dia = jueves;
    printf ("\n Cuarto día de la semana: %d", dia);
    dia = viernes;
    printf ("\n Quinto día de la semana: %d", dia);
    dia = sabado;
    printf ("\n Sexto día de la semana: %d", dia);
    dia = domingo;
    printf ("\n Ultimo día de la semana: %d", dia);
}

```



Es interesante resaltar que no hay formato de impresión para tipos enumerados. Se proyectan sobre sus valores enteros.

Cada uno de los elementos de la enumeración lleva asociado un valor entero. Por defecto, el primer elemento tiene asociado el valor 0, el segundo el valor 1 y así sucesivamente. De acuerdo a esto, **sabado** es igual al valor 5.

También es posible asignar un valor distinto a cada uno de los elementos de la enumeración. A continuación se muestra un ejemplo en el que se define el tipo enumerado **Respuesta**, que puede tomar dos valores: **No** con valor -1 y **Si** con valor 1.

```
enum Respuesta {No = -1, Si = 1};
```

En C se puede recurrir a los tipos enumerados para definir el tipo de datos booleano de la siguiente forma:

```
enum Boolean {false, true};
```

De esta forma, **false** será tratado como falso (valor 0) y **true** como verdadero (valor 1).

## 4.5 Tipos avanzados

A partir de los tipos básicos descritos, se definen en C tipos avanzados compuestos por colecciones o agrupaciones de elementos de tipos básicos. Los tipos avanzados son los arrays, cadenas de caracteres, estructuras y punteros. Aunque estos tipos se van a estudiar con más detalle en capítulos posteriores, se presentan en esta sección unos breves ejemplos de los mismos.

### 4.5.1 Arrays

Un array es un conjunto de datos del mismo tipo, a los que se da un nombre común y que se almacenan en posiciones de memoria contiguas. A un elemento específico de un array se accede mediante un índice, que siempre empieza en el elemento